Contract based management of the memory resource

Ismael Ripoll, Patricia Ballastre, Miguel Masmano, Alfons Crespo, Alan Burns

In Proc. of the 17th International Conference on Real-Time and Network Systems
RTNS'2009, Paris, ECE, 26-27 October, 2009

Contract based management of the memory resource

Ismael Ripoll[†], Patricia Balbastre[†], Miguel Masmano[†], Alfons Crespo[†], and Alan Burns[‡*]
† Instituto de Informática Industrial, Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022 Valencia, Spain
‡ Deparment of Computer Science
University of York, Heslington, York, YO10 5DD, UK

Abstract

Resource reservation has been widely used in many real-time systems to guarantee the proper access to the system resources. Despite that being the memory a key resource, it has attracted little attention in the specific area of real-time systems. In order to use dynamic memory in real-time systems, two fundamental problems have to be settled: allocation and deallocation in bounded time, and the fragmentation problem.

Recent research results have removed the unbounded timing behaviour of the dynamic memory allocation. TLSF is a fast and constant time memory allocator. Although the fragmentation is still an open research problem, we present a deep and comparative analysis showing that it has several characteristics in common with the well-known WCET analysis.

In this paper, we present a contract based framework for handling dynamic memory in real-time systems. The framework provides both: i) timing guarantee for dynamic memory allocation and deallocation operations, and ii) spatial guarantee by using a flexible contract negotiation model.

1 Introduction

Nowadays, computers are included as components in many kinds of systems. We can find them in automotive, aerospace, robotics, home systems, toys, etc. Most of these embedded systems need to interact with the environment and operate under real-time constraints. Moreover, the wide field of applications requires an important engineering and programming effort to add flexibility and adaptability to these systems. To fulfil these requirements the complexity of embedded software increases.

A basic property that differentiates embedded systems from other general software systems is their execution environment under resource constraints. These resource-constrained applications are related to CPU time, memory usage, I/O, network bandwidth, energy, space, etc.

Resource-aware computing tackles this challenge by managing them dynamically, and thus, optimising the use of these resources. Resource managers are the software components responsible for performing this task. In real-time systems, some resource-management related topics such as CPU time deliverance and the relationship between CPU and power consumption have been deeply studied providing consolidated theoretical foundations. However, other topics such as memory management have been completely ignored. Like CPU, memory is a fundamental part of the embedded system, nevertheless, real-time systems allocates statically the existing memory at load time, most of the times, overestimating the real necessities in memory of the applications. However, actual applications seldom behaves statically but they are usually composed by a set of static regions (code and data) which do not vary during the program's execution and dynamic ones (stack and heap) which grow/shrink according to application needs. The stack can be considered as a limited region not being particularly affected by a static deliverance of the memory, however, the absence of a heap prevents applications to use dynamic structures. As a direct consequence of this, embedded systems designs tend to require more memory than they really would need if this resource was dynamically allocated.

Moreover, the problem of resource management is that different resources cannot be handled independently. Currently real-time operating systems are including some sort of resource manager. This is an emerging trend, and more research on techniques and mechanisms to use and control the system resources (application isolation) are needed.

This work is motivated by the necessity of including memory resource management in a real-time resource management framework (FRESCOR). Dynamic storage allocation is considered one of the keys to add flexibility and adaptability to the application programming. For this rea-

*This work was supported by FRESCOR and the Spanish Government Research Office (CICYT) under grant THREAD (TIC2005-08665-C03)

son, it becomes increasingly desirable. However, dynamic memory has seldom been used in real-time applications due to its unbounded nature.

Recently, a new real-time allocator (TLSF, Two-Level Segregate Fit)[16, 14] was proposed specifically designed to meet real-time requirements. This is the first allocator able to perform allocation/deallocation in constant-time, O(1), with a very high efficiency in term of time and space (fragmentation). The use of this allocator opens new options to consider dynamic storage allocation in real-time applications due to its bounded response time.

Although, the proposed model is general and could cover the memory of the programs in execution, we have only considered the memory usage related to dynamic memory. Static memory as a unique resource is a well-known bin-packing problem. When memory is not static or more than one resource is considered, the problem is not so trivial. Static memory jointly with CPU allocation has been considered in [7]. It proposes a way to analyse, in a multiprocessor platform with limited resources, the computing capacity at each processor and the amount of local memory for a set of tasks.

The approach presented in this paper is being developed in the FRESCOR project [8] where other resources such as CPU and network are being developed by other research groups which will be integrated at the last stages of the project.

1.1 Contributions and outline

In this paper we propose a memory resource reservation model for flexible embedded systems requiring dynamic memory. The static memory to execute the applications (program code, static data and stack) is not considered in this work.

The main contributions of this paper are:

- A vision of the memory as a resource in the same way that other resources are considered.
- A memory model and a memory reservation architecture being able to manage the spare memory of the system.
- An acceptance test for memory contracts and a memory reclaiming mechanism of the memory associated requests.

As far as the authors know, this is the first work dealing with quality of service related to memory management of the application.

This paper is organised as follows: section 2 presents the most common misconceptions about dynamic memory in real-time systems. In section 3 the parallelism between CPU and memory management is established. In section 4, the memory model is presented. Section 5 characterises the states of the memory and proposes an acceptance test for dynamic memory requests. The proposed acceptance test is evaluated in section 6. Section 7 summarises the existing works on resource reservation and dynamic memory in real-time systems. Finally, section 8 presents a summary of the results of this work and outlines future work on this topic.

2 Dynamic memory misconceptions

2.1 Bounded time allocation

Before the TLSF allocator [**?**] was presented, the use of dynamic storage allocation (DSA) was avoided in hard real-time applications. In [18] and [6], authors provide arguments to avoid its use due to the unbounded operations.

The only allocator with a bounded operation cost used in real-time applications was the well known Binary-buddy (or any other of the "buddy" family allocators). Being H the size of the heap, this allocator has an $O(log_2(H))$ cost on both, allocation and deallocation, but causes a large internal fragmentation (close to 50%).

Another generally accepted idea [4] is that there is a trade-off between space and time efficient allocators. In other words, a fast and bounded allocation can only be achieved at the cost of increased wasted memory due to "fragmentation".

These misconceptions were analysed by Johnstone et al. [11], and concluded that **it is possible to design space efficient allocators using well known allocation policies** (best-fit or good-fit), and those policies can be implemented using fast mechanism (segregated lists).

The publication of the TLSF allocator changed some of the assumptions taken from granted about how a DSA works. The TLSF memory allocator implements all dynamic memory operations (malloc, free, realloc) in constant time regarding the stated of the pool. The performance of the TLSF is not affected by the fragmentation or by the amount of different free blocks sizes. Also, the average time of the TLSF is close to the fastest allocators (DLmalloc[12]).

2.2 Memory fragmentation

P. Wilson et al. [23] define the fragmentation as *the inability to reuse memory that is free*. This definition focuses only of the final result of the allocation and deallocation process. It is also interesting to note that fragmentation depends both, on the current memory state, and on the future application requests.

The fragmentation problem was largely studied since the very beginning of the computer science. But, in many cases

the results of a researcher fell in contradiction with the previous ones. Zorn et al. [24] discovered a very important fact when defining a standard set of synthetic workload for evaluating DSAs. They tried to define a reference set of synthetic models which reproduce the allocation/deallocation sequence behaviour of a group of selected applications. They worked with both, models that were used by other researchers, and original models. Even the most accurate and complex mathematical model they were able to roughly approximate the behaviour of real program. The main conclusion was that most of the previous results should be reconsidered or even invalidated.

M.S. Johnstone and P.R. Wilson[11] analysed real applications and current policies and concluded that the fragmentation "problem" is really a problem of poor allocator *implementations*, and that for these programs well-known policies suffer from almost no true fragmentation. In addition, very good implementations of the best policies are already known.

We agree with the ideas of M.S. Johnstone and P.R. Wilson [11] about the nature and real impact of the fragmentation. Therefore, **the fragmentation problem can be bounded in most real applications, which effectively enables the use of dynamic memory**. In any case, further research should be carried out to fully understand and seize the problem, using formal and analytical methods rather than simulations and practical experiments.

The TLSF allocator follows all the policies shown by M.S. Johnstone et al. which reduce fragmentation (immediate coalescing, good fit and reuse blocks which has been releases recently). Those policies were implemented using a clever set of segregated list ranges which both, can be implemented using O(1) algorithms and causes only a 3% worst case internal fragmentation.

3 Processor versus memory

Fragmentation vs. execution time: The memory fragmentation problem has many similarities with the WCET analysis. In both cases, the analytical estimation of the worst case value is hard to find and quite pessimistic.

Currently, the worst case fragmentation (WCF) is only known for small set of allocation policies [20]: best-fit and first-fit. As far as the authors know, there are little WCF analysis about the family of allocation policies known as good-fit, which are ones used on the allocators that also show constant time operation (TLSF and Half-fit). Both, the conclusions about the practical fragmentation of Johnstone at al. [11], and the fact that external fragmentation can be reduced by increasing internal one [19][1], make us optimistic about the possibility to see advances in the WCF for real-time systems.

Physically allocatable resources: In a conventional system, a process can be preempted and suspended for a long time. When later resumed, the final logical result will still be correct. Also, the processor can be run as long as required to solve a given problem (tasks do not have deadlines). In a conventional system, the processor is a very flexible and dynamic resource.

One of the main differences between conventional and real-time systems is that the processor is managed as a statically allocated resource. The real-time scheduler is responsible of allocating to each task the amount of resources, $\frac{C_i}{P_i}$, granted to it. On a periodic system, a fraction of the total processor time is allocated to each task. The periodic nature of most real-time systems, jointly with the scheduler, transforms the processor into a bounded resource: the processor cannot be over 100% utilisation (or less depending on the scheduling policy).

More similarities can be found when comparing the memory management with the CPU reservation based servers. For instance, a Periodic Server is invoked with a fixed period to spend a server's capacity. This capacity is consumed by the tasks ready to use or, if there are not tasks ready, it is idled away. So, the capacity is preserved.

The memory manager has a capacity (maximum amount of memory) that it can provide to its associated tasks. This capacity persists along its execution and never is replenished (renewed). From this point of view, it is a *non-renewable resource*. The application, or the associated tasks, have the responsablity of usign the resource in the proper way to allocate memory blocks and freeing them.

Feasibility analysis: Contrarily to the processor feasibility analysis, the analysis of the total amount of memory required to run an application is quite straightforward. Considering that the application does not use dynamic memory, the total application required memory can be obtained from the compiler and estimating the amount of stack memory[2] used. These two tests, for processor and memory, are done off-line during the development phase.

Spare capacity: In most cases, the system has more resources than those strictly needed by the application: the processor is not fully utilised and there are free memory. A lot of research has been done trying to take advantage of the processor spare capacity by using aperiodic and bandwidth servers. The basic idea is to use the extra capacity to improve the quality of the system response but without jeop-

[1] For example, an extreme case occurs when the allocator rounds-up all block requests to a one single size. In this case, no external fragmentation happens.

[2] Which basically depends on the number of nested function calls and the local variables.

ardising the correct execution of the hard real-time tasks. A server is an abstract entity used by the scheduler to reserve a fraction of CPU-time to a task.

4 Memory model

This section details the memory model and the underlying architecture which supports the memory management.

4.1 Memory resource manager

The dynamic memory pool is managed by the memory resource manager (MRM) that is the layer in charge of negotiating the memory reservation requests (*contracts*) between applications and the system. The contract has to be negotiated between both, application and MRM, to guarantee the availability of the resources. As a consequence of a successful negotiation, the MRM creates a memory resource controller (R)[3] that will monitor and control the use of the resource. The application, as owner of the resource, binds one or more tasks to it. An application can negotiate several contracts with the MRM. Also, a task can be bounded to several granted resources.

The MRM is defined as a set of resources $\Upsilon = \{R^1, R^2, \cdots, R^n\}$, and a memory pool $\Omega = (M_{tot}, M_{FR}, M_{CR})$.

4.2 Memory resources

A memory resource controller (R^k) is the component that monitors and controls the use of the granted memory. Each R is characterised by the following parameters:

$$R^k = (R^k_{min}, R^k_{max}, R^k_{imp}, R^k_{stab}, R^k_{at}, R^k_B, R^k_{clive}, R^k_{mlive})$$

where R^k_{min} R^k_{max} are the minimum and maximum amount of memory that the task needs to operate properly. The value of the granted budget will be in the range of these values; R^k_{imp} is the absolute fixed importance; R^k_{stab} is the duration relative to the time at which the request is made, during which the memory assigned to the application must not change, R^k_{at} specifies the arrival time of the negotiated and accepted contract, R^k_B is the budget of memory granted in the negotiation; R^k_{clive} is the *live memory* as the sum of all the currently allocated memory blocks to this resource and R^k_{mlive} is the maximum value reached by the current live memory.

The first four parameters are provided in the contract negotiation, the rest are required by the resource controller to monitor and control the resource usage.

[3]It corresponds to a *virtual resource* controller but we avoid the use of the term *virtual* because of virtual memory has a well-known meaning in memory management

In case that an application does not require dynamic memory, the only parameter needed by the MRM is R_{min} which corresponds to the static amount of memory needed by the application. As a consequence, $R^k_B = R^k_{min}$, and the rest of parameters are not required. Note that the static memory model is a subset of our proposal.

The R_{max} parameter can be estimated from the dynamic data needed by the application. If the dynamic memory is stored in a buffer until a second thread remove it, it can be considered that the minimal application requirements are achieved with a minimum buffer size, but the larger buffer size the better results are obtained. The R_{max} (R_{min}) parameter will be the sum of the static memory and the maximum (minimum) buffer size.

A resource R^i is said to be *eligible* at time t_{now} if its stability time has expired: $R^i_{at} + R^i_{stab} \leq t_{now}$.

Let Υ^k_- be the subset of elegible resources with lower importance than R^k:

$$\Upsilon^k_- = \left\{R^i \ / \ \left(R^i_{imp} < R^k_{imp}\right) \wedge \left(R^i_{at} + R^i_{stab} \leq t_{now}\right)\right\}$$

Similarly, Υ^k_+ is the subset of elegible resources with higher importance than R^k:

$$\Upsilon^k_+ = \left\{R^i \ / \ \left(R^i_{imp} > R^k_{imp}\right) \wedge \left(R^i_{at} + R^i_{stab} \leq t_{now}\right)\right\}$$

4.3 Memory pool characterisation

The memory pool is characterised by three parameters $\Omega = (M_{tot}, M_{FR}, M_{CR})$. M_{tot} is the total amount of "heap" system memory (memory used to attend dynamic memory requests). All the dynamic memory used by applications will be managed using a single memory pool. In other words, when a resource is created, no memory is reserved (or allocated) from the heap to create a new subheap. The resource operates as a proxy to the single system pool, implements the access policy and enforces strict resource isolation, by accounting how much memory was granted initially, and how much memory can still be requested to it. M_{tot} is not entirely available for applications. Some amount of this memory will be reserved for different purposes. Specifically:

$$M_{tot} = M_u + M_{FR}$$

where:

M_u is the effective memory guaranteed to be used by the dynamic storage allocation.

M_{FR} is the memory reserved to deal with fragmentation. This parameter mainly depends on two factors: the efficiency of the dynamic memory allocator and the behaviour of the application (mutator[4]). As explained

[4]The term "*mutator*" is used in the area of dynamic memory to refer to the application that uses (allocates and frees) memory.

in section 2.2, there are some allocating policies that are known to produce less fragmentation; the TLSF allocator has been shown to be among the most efficient allocators. Experimental results [16, 14] show that no real application has caused more than 15% of total fragmentation (both internal and external). As a rule of thumb, we suggest to reserve $FR = 20\%$ of the memory, so $M_{FR} = \frac{M_u \cdot FR}{100}$. In any case, it is recommended to analyse the exact memory needs of the applications and adjust this parameter accordingly.

A fraction of the M_u can be reserved to be used when the system is under a high demand of memory, to attend new incoming contract negotiations. Let CR be the percentage of memory reserved for run time contract negotiation. For convenience, let $M_{CR} = \frac{M_u \cdot CR}{100}$.

At any time, the remaining memory M_r that can be granted to incoming contracts can be calculated as:

$$M_r = M_u - \sum_{1 \leq i \leq n} R_B^i - M_{CR}$$

Negative values of M_r mean that the system is using memory from M_{CR} and, consequently, it is in a stressed situation. Figure 1 shows graphically how the memory pool is managed.

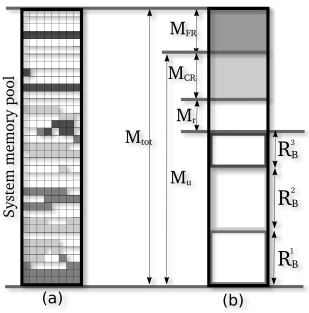

Figure 1. Memory pool parameters.

Figure 1.*a)* represents the memory pool with the used and free blocks distributed all along the memory. Figure 1.*b)* draws the memory abstraction as it is offered by the MRM.

4.4 Memory access protocol

In order to use dynamic memory, the application has to follow a three-step protocol (sketched in Figure 2): negotiation, binding and usage.

In the first step, the application negotiates the use of the resource (contract). If it succeeds, the memory resource controller R^k is created and the application tasks can be associated to it. After the binding step, the tasks can use the

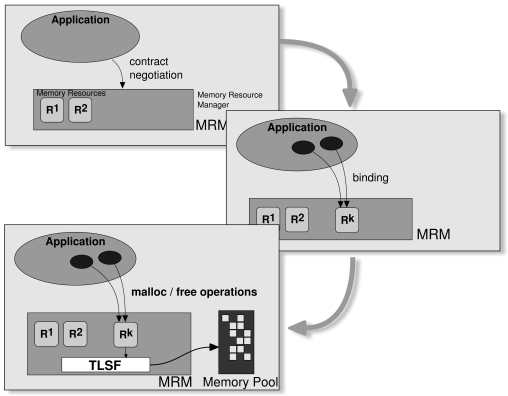

Figure 2. Memory access protocol

memory allocating (malloc) and deallocating (free) operations. These operations are redirected to the memory allocator (TLSF) which receives the invocations of all memory resource controllers. TLSF uses a unique memory pool for all these memory allocations.

The goal of the contract negotiation is to determine the amount of memory granted to the resource, that is, its budget (R_B^k). The contract negotiation will succeed if $R_{min}^k \leq R_B^k \leq R_{max}^k$.

Three aspects have to be considered during the negotiation: i) the amount of memory to be assigned, ii) which R^i are eligible for reducing their budgets and iii) how much memory can be reclaimed from each one. Follows a formalisation of these ideas, which will be used in the acceptance test.

The amount of memory to be assigned will be the maximum requested when there is enough memory or it can be obtained from less important memory resources. If more important memory resources are reclaimed, then the *goal* is to satisfy the minimum memory requested.

The R^i candidates to reduce its budget are the eligible ones. Regarding the amount of memory reclaimed from each R^i, tree levels of aggressiveness reclaiming can be defined:

1. Memory that never has been used. It corresponds to the difference between the budget and the maximum live memory: $R_B^i - \max\{R_{min}^i, R_{mlive}^i\}$

2. Memory that is not currently being used. It corresponds to the difference between the budget and the current live memory: $R_B^i - \max\{R_{min}^i, R_{clive}^i\}$.

3. All the extra memory. The difference between the budget and its minimum: $R_B^i - R_{min}^i$.

Figure 3 shows an example of the dynamic memory evolution in a memory resource (R^k). It plots the amount of

memory allocated by R^k as result of dynamic memory allocations and deallocations. At the end of the stabilisation time, this resource is eligible for reclamation. At t_{now}, a new contract is under negotiation. At this time, the three reclamation levels are drawn in the figure.

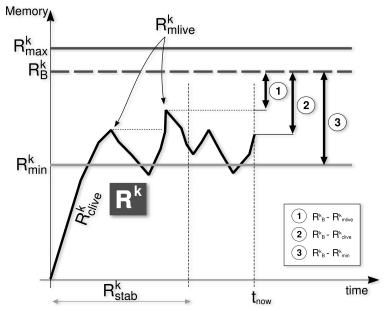

Figure 3. Dynamic memory used by R

If the third level is applied, then the current live memory may be larger than the new budget. There are several actions that can be done to overcome this transient overload situation:

Do nothing: But no allocation can be done (only deallocation) until the live memory will be below the budget. The system will be in an unsafe state until the live memory is reduced, which is not acceptable on a real-time system.

Inform the application: Ask to the application to deallocate memory. As the application has the knowledge about the semantics of the data, it can properly deallocate the most convenient blocks.

Deallocate memory: The MRM releases the last allocated blocks. This is not acceptable, because it breaks the basic rules of the program execution.

In this paper, only the two first levels will be considered for memory reclaiming.

5 Acceptance test

5.1 Characterisation of the memory states

In this section, we analyse the reachable states of the memory taking into account the contract requirements, the current memory status and the reclaiming capacity applied. Figure 4 shows different cases that can occur comparing the available and the requested memory.

In case 1 there is enough available memory to fulfil the request needs, so the memory requested is granted. Cases 2, 3 and 4 perform a memory reservation request that cannot

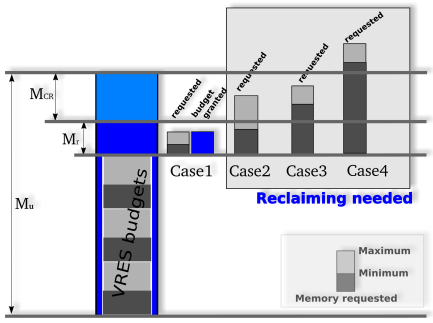

Figure 4. Negotiation with space memory attributes.

be granted directly because it involves not only the remaining memory but the reserved memory (M_{CR}) or exceeds the memory capacity. These cases require a reclaiming process on other memory resources in order to recover memory that is not used.

Figure 5 shows the three initial situations or states (cases 2, 3, and 4) and the final memory states reachable depending on the amount of memory reclaimed to the other resources.

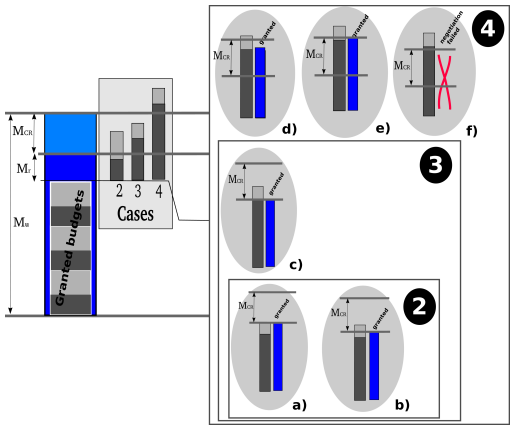

Figure 5. Final states after the reclamation phase for cases 2, 3 and 4

a) During the reclaiming phase from lower important resources, the memory available reaches the maximum requested ($M_r > R^k_{max}$). It is granted (Goal = R^k_{max}).

b) The reclaimed memory to lower or equal important resources is enough to grant the minimum ($M_r > R^k_{min}$) but not sufficient to grant the maximum ($M_r < R^k_{max}$). Before reducing to higher important resources, this amount is granted. (Goal = $[R^k_{min}, R^k_{max}]$)

c) The reclaimed memory is the minimum requested reducing memory from lower, equal and higher important

resources ($M_r = R^k_{min}$). (Goal = R^k_{min}).

d) After the reclaiming phase, it could not obtain enough memory ($M_r > R^k_{min} > M_r + M_{CR}$). To grant the minimum memory requested, it uses the reserved memory.

e) After the most aggressive reclaiming phase, the amount of memory obtained was the minimum using all the available (remaining + reserved).

f) After the reclaiming phase, it could not obtain enough memory ($M_r + M_{CR} < R^k_{min}$). In this case, the memory contract cannot be granted and the **negotiation fails**.

Case 2 can reach states *a)* and *b)*. Case 3 can additionally arrive to state *c)*. Finally, Case 4 can reach any of the states.

5.2 Acceptance algorithm

The reclaiming phase is in charge of recovering memory already assigned to an existing resource. It is easy to test if there is enough memory: if $R^k_{max} \leq M_r$, the negotiation succeeds and the granted budget is R^k_{max}. If not, it will reclaim memory from other resources, and the budget goal will be reduced as the MRM reclaims from more resources. The goal is that a resource cannot have its maximum memory if higher important resources do not have it. The reclaiming process consists of four reclaiming steps. Listing 1 sketches the actions done during the contract negotiation (the functions that carry out the reclamation are explained below).

Listing 1. Pseudo-code of the acceptance test

```
1  function acceptance_memory_test(R^k) is
2  begin
3      -- Phase 0: Use remaining memory.
4      --          Budget goal is : R^k_max
5      if (M_r ≥ R^k_max) then
6          return(GRANTED);
7      end if ;
8      -- Phase 1: Reclaim memory never used from lower
9      -- imp resources . Budget goal is : R^k_max
10     M_r += reclaim_memory_from_ϒ^k_−(R^k);
11     if (M_r ≥ R^k_max) then
12         return(GRANTED);
13     elsif (M_r > R^k_min)
14       return(GRANTED);
15     end if ;
16     -- Phase 2: Reclaim memory never used from higher
17     -- imp resources. Budget goal is : R^k_min
18     M_r += reclaim_memory_from_ϒ^k_+(R^k);
19     if (M_r ≥ R^k_min) then
20         return(GRANTED);
21     end if ;
22     -- Phase 3: Use reserved memory.
23     -- Budget goal is: R^k_min
24     M_r += M_CR;
25     if (M_r ≥ R^k_min) then
26         return(GRANTED);
27     end if ;
28     -- Phase 4: Reclaim memory not currently being
29     -- used from lower imp resources .
30     -- Budget goal is: R^k_min
31     M_r += reclaim_live_memory_from_ϒ^k_−(R^k);
32     if (M_r ≥ R^k_min) then
33         return(GRANTED);
34     end if ;
35     return(FAILED);
36 end acceptance_memory_test;
```

This test performs an analysis of the memory state and determines whether the memory requested can be granted or not. The reclaiming phase is performed with different levels of aggressiveness or depth.

During the first two phases, the system is considered to be unstressed because 1) there is free memory not allocated to any resource or; 2) because the memory previously reserved to less important resources has been never used by those resources, therefore we assume that the application made an overestimation of the maximum memory, and that memory can be reclaimed with no damage.

If the algorithm cannot attend the new contract using the memory obtained from the first two phases, then we consider that the system is stressed, or it is close to. In this case, the amount of memory that will be given to the new resource R^k is not longer the maximum requested, but the minimum.

A particular case may occur if at the end of phase 1 there is not enough memory to attend the maximum memory requested, but there is more memory than the minimum requested. In this case, the new budget will be a value in the range $[R^k_{min}..R^k_{max}]$.

This algorithm ends when enough memory is found or after the fourth phase. If the algorithm ends successfully, then the variable M_r contains, at least, the minimum memory requested in the new contract. In this case, the new resource is created, and the remaining memory as well as the budget of all affected resources are updated accordingly. If the algorithm fails (not enough memory), the state of the resources remains unchanged. It is implemented as a transaction: system and resource data are copied in a scratchpad area, the acceptance test works on the scratchpad area, and then, only if the algorithm ends successfully the results are committed to the real system and resource data. Otherwise, the scratchpad data is discarded.

The next pseudo-code details the reclamation phases 1, 2 and 4:

```
1  procedure reclaim_memory_from_Υ⁻ᵏ(Rᵏ) is
2      foreach Rⁱ ∈ Υ⁻ⁱ
3          M_r+ = R^i_B − max(R^i_mlive, R^i_min);
4          exit when M_r ≥ R^k_max;
5      end for
6  end reclaim_memory_from_Υ⁻ᵏ;
```

```
1  procedure reclaim_memory_from_Υ⁺ᵏ(Rᵏ) is
2      foreach Rⁱ ∈ Υ⁺ⁱ
3          M_r+ = R^i_B − max(R^i_mlive, R^i_min);
4          exit when M_r ≥ R^k_min;
5      end for
6  end reclaim_memory_from_Υ⁻ᵏ;
```

```
1  procedure reclaim_live_memory_from_Υ⁻ᵏ(Rᵏ) is
2      foreach Rⁱ ∈ Υ⁻ⁱ
3          M_r+ = R^i_B − max(R^i_clive, R^i_min);
4          exit when M_r ≥ R^k_min;
5      end for
6  end reclaim_live_memory_from_Υ⁻ᵏ;
```

6 Evaluation

This section presents the evaluation of the memory resource controller. Next sections detail the evolution of two scenarios when different resources are defined. These scenarios are representatives of the results obtained with different set of resources. The scenarios presented in this section intend to show how the algoritm work and the evolution of the memory assigned to the different resources. It is not relevant the sizes showed in the scenario ($M_u = 6000$), that is relevant is the relation between the total amount of memory and the memory requested by the resources. Finally, an evaluation of different sets of scenarios is presented.

6.1 Scenario 1

This scenario shows the memory negotiated by five resources with usable memory of $M_u = 6000$ bytes and a memory reservation of the 15% (M_{CR}= 900). Each resource R^i has the following characteristics:

	R^k_{at}	R^k_{imp}	R^k_{min}	R^k_{max}	R^k_{stab}
R^1	10	1	1500	1800	400
R^2	10	5	800	1400	300
R^3	450	3	500	700	300
R^4	600	4	500	1400	300
R^5	950	2	500	800	300

This scenario produces a result that is plot in figure 6. In this figure, the granted budget (straight line) and the evolution of the mallocs and frees is plotted for each resource. As it can be seen, no reclamation is needed for resources R^1, R^2 and R^3 because there is enough memory for granting the maximum amount requested. However, at time 600, there is not enough memory to satisfy R^4 request. Specifically, this situation corresponds to case 2 (figure 5). When R^4 arrives at time 600, R^1 and R^2 are eligible. As R^2 has a higher importance (5) than R^4 (4) and the memory available is not enough (100 bytes remaining, 900 bytes reserved) the algorithm enters in phase 1 (reclaim_memory_from_Υ^k_-). In this case, the budget of R^1 (lower importance) is reduced (R^1 decreases from 1800 to 1500). When R^5 arrives, it requests its maximum memory (800), which cannot be granted with the remaining memory. This situation corresponds with case 3 (figure 5), since $M_r + M_{CR}$ does not grant the R^5_{max} but do grant R^5_{min}. The reclamation phase tries to recover memory from phase 1 (at time 950, only R^1 is an eligible resource with lower importance than R^5). However, R^1 already has decreased its budget in a previous reclamation (to grant memory to R^4 at time 600), so a second reclamation phase is executed (reclaim_memory_from_Υ^k_+). After this phase, the minimum memory can be granted to R^5. Note that R^5 could not obtain its maximum memory request because it implies to reduce the amount of memory to higher importance resources.

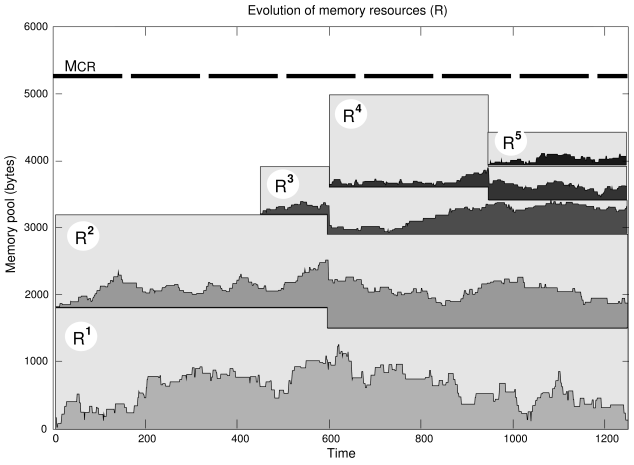

Figure 6. Scenario 1.

6.2 Scenario 2

In this scenario, R^5 has greater maximum and minimum requested memory so the total minimum memory of all resources is higher than the total usable memory. This situation corresponds to case 4 (figure 5). The graphical response of this scenario is plot in figure 7. This scenario

coincides with scenario 1 until time 950. At this time there is no remaining memory and a reclamation phase is started. However, the reclaimed memory from R^2, R^3 and R^4) is not enough to fulfil the request and the algorithm enters in phase 3 of the acceptance test, using 150 bytes of the reserved memory (M_{CR}). Note that using the reserved memory implies to assign the minimum memory.

	R^k_{at}	R^k_{imp}	R^k_{min}	R^k_{max}	R^k_{stab}
R^5	950	2	1950	2500	300

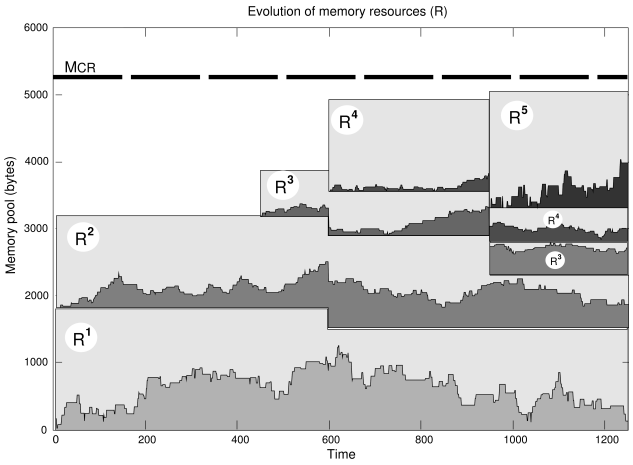

Figure 7. Scenario 2.

7 Related work

Resource reservation in real-time systems is a topic that has received the attention of researchers for more than twenty years. When considering the CPU usage, resource-based algorithms have been developed to characterise the timing requirements and processor capacity reservation requirements for real-time applications ([17, 10, 9, 3, 1, 2]).

In [22], the authors pointed out the needs of resource reservation in many application domains, such as, aerospace, multimedia, and real-time control systems. They stated a set of propositions related to the operating systems services to provide quality of service to applications. They also claimed for a more efficient memory management and a definition of the memory services interfaces at RTOS level.

The problem of task partitioning on multiprocessors, considering both CPU and memory as a resource, is formalised in [7]. However, the memory is considered as a static resource, so tasks cannot vary their memory assignment. This work proposes techniques for simultaneously considering constraints due to several resources: the computing capacity at each processor, and the amount of local memory available.

As far as the authors know, the first paper considering the dynamic memory management in real-time systems is [13]. In this paper, it is proposed to control the memory allocation and deallocations considering that tasks request memory in a periodic fashion. A feasibility test for memory is also proposed. This paper does not defined any memory reclamation and adjustment algorithm. The control of the memory was based on the knowledge about the periodic allocation and deallocation performed by the tasks.

In [21] a memory reservation mechanism (container) to monitor and control the use of resources (CPU time and resident memory) in Linux systems is proposed. The proposed mechanism isolates the memory behaviour of a group of tasks from the rest of the system. It can be used to isolate greedy applications by limiting the amount of memory, execution of virtual machines, etc. This mechanism will be included in the new version of the Linux kernel.

Our proposal could have similarities with the elastic task model proposed by Buttazzo et al [5]. This paper states that: *"whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilisation of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request."*. The MRM reclaims and reduces the granted memory to existing resources in order to accommodate a new incoming resource.

8 Conclusion and future work

This paper presents a novel memory reservation framework, jointly with an acceptance test that redistributes the available memory to improve the overall system performance.

The use of dynamic memory in real-time systems was very limited due to the unbounded nature of the basic allocation and deallocation operations. The situation changed when the TLSF algorithm was presented by Masmano et al. [15]. The TLSF is a fast constant time, $O(1)$, allocator.

The other source of indeterminism, which seriously limits the use of DSA in real-time, is the memory fragmentation problem. We summarise the main results about fragmentation, and conclude that although it is still an open problem, it is not a real problem for practical applications. The fragmentation problem is comparable with the worst case execution time (WCET) analysis. In both cases, the theoretical worst case are very pessimistic compared with the real observed fragmentation

Contrary to general belief, a detailed comparison between how the processor is scheduled and how the memory can be used in real-time systems, shows that both kinds of resources have more similarities than differences.

In the second part of the paper, a memory model for real-time applications, and a contract based framework for

managing spare memory is presented. The proposed acceptance test resembles the elastic task model, in the sense that the acceptance test distributes the spare memory among the memory resource controller that can use it. Different reclaiming memory strategies are analysed and used to adjust the available resources.

References

[1] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Journal of Real-Time Systems*, 29(2-3):131–155, 2005.

[2] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, J. J. Gutiérrez, T. Lennvall, G. Lipari, J. M. Martínez, J. L. Medina, J. C. P. Gutiérrez, and M. Trimarchi. Fsf: A real-time scheduling architecture framework. In *IEEE Real Time Technology and Applications Symposium*, pages 113–124, 2006.

[3] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems*, 22(1-2):49–75, 2002.

[4] A. Borg, A. Wellings, C. Gill, and R. K. Cytron. Real-time memory management: Life and times. *Euromicro Conference on Real-Time Systems*, 0:237–250, 2006.

[5] G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *IEEE Real-Time Systems Symposium*, pages 286–295, December 1998.

[6] S. Feizabadi. Dynamic memory management in a resource-constrained real-time utility accrual environment. In *PhD Dissertation Proposal*, 2004.

[7] N. Fisher, J. H. Anderson, and S. Baruah. Task partitioning upon memory-constrained multiprocessors. In *IEEE Real Time Technology and Applications Symposium*, 2005.

[8] FRESCOR. Framework for Real-time Embedded Systems based on COntRacts, 2007. FP6/2005/IST/5-034026 European Research Project. (http://www.frescor.org).

[9] C. Hamann, J. Loser, L. Reuther, S. Schonberg, J. Wolter, and H. Hartig. Quality-assuring scheduling: Using stochastic behavior to improve resource utilization. In *22nd IEEE Real-Time Systems Symposium*, pages 119–128, 2001.

[10] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 480–491, 1998.

[11] M. Johnstone and P. Wilson. The Memory Fragmentation Problem: Solved ? In *Proc. of the Int. Symposium on Memory Management (ISMM'98), Vancouver, Canada*. ACM Press, 1998.

[12] D. Lea. A Memory Allocator. *Unix/Mail*, 6/96, 1996.

[13] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory resource management for real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 201–210, 2007.

[14] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo. A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, 40(2):149–179, 2008.

[15] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *16th ECRTS*, pages 79–88, Catania, Italy, July 2004. IEEE.

[16] M. Masmano, I. Ripoll, J. Real, A. Crespo, and A. J. Wellings. Implementation of a constant-time dynamic storage allocator. *Softw., Pract. Exper.*, 38(10):995–1026, 2008.

[17] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical report, Pittsburgh, PA, USA, 1993.

[18] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. In *14th ECRTS*, page 41, 2002.

[19] B. Randell. A Note on Storage Fragmentation and Program Segmentation. *Communications of the ACM*, 12(7):365–372, 1969.

[20] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 1977.

[21] B. Singh and V. Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Linux Symposium*, 2007.

[22] L. Steffens, G. Fohler, G. Lipari, and G. Buttazzo. Resource reservation in real-time operating systems - a joint industrial and academic position. In *ARTOSS'03*, pages 25–30, 2003.

[23] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Int. Workshop on Memory Management*, volume 986 of *LNCS*, pages 1–16. Springer-Verlag, 1995.

[24] B. Zorn and D. Grunwald. Evaluating Models of Memory Allocation. *ACM Transactions on Modeling and Computer Simulation*, pages 107–131, 1994.